Edgebreaker: Compressing the incidence graph of triangle meshes

Jarek Rossignac
GVU Center, Georgia Institute of Technology
http://www.gvu.gatech.edu/people/faculty/jarek.rossignac/

May 22, 1998 (Revised June 16, 1998)

Abstract

Edgebreaker is a simple scheme for compressing the triangle/vertex incidence graphs (sometimes called topology) of three-dimensional triangle meshes. Edgebreaker improves upon the worst case and the expected compression ratios of previously reported schemes, most of which require O(nlogn) bits to store the incidence graph of a mesh of n triangles. Edgebreaker requires only 2n bits or less for simple meshes and can also support fully general meshes by using additional storage per handle and hole. Edgebreaker's compression and decompression processes perform an identical traversal of the mesh from one triangle to an adjacent one. At each stage, compression produces an op-code describing the topological relation between the current triangle and the boundary of the remaining part of the mesh. Decompression uses these op-codes to reconstruct the entire incidence graph. Because Edgebreaker's compression and decompression are independent of the vertex locations, Edgebreaker may be combined with a variety of vertex-compressing techniques that exploit topological information about the mesh to better estimate vertex locations. Edgebreaker may be used to transfer the entire surface bounding a 3D polyhedron or only a triangulated surface patch, for which the bounding loops are already known and need not be transferred. Its superior compression capabilities, the simplicity of its implementation, and its versatility make Edgebreaker the perfect candidate for the emerging 3D data exchange standards for interactive graphic applications. The paper sets geometric compression in a formal topological framework and offers a new comparative perspective on prior art.

A. Introduction

A.1. Problem statement and contribution summary

The objective of the technique reported here is to compress the incidence graph (sometimes called topology) of a triangle-mesh without exploiting any knowledge of the location of its vertices. In our scheme, called Edgebreaker, the compression and the decompression processes "invade" the mesh by "breaking through" one edge (called *gate*) of its boundary at a time and by removing (compression) or recreating (decompression) the triangle incident upon the gate.

Edgebreaker can encode, with $2t+b-2$ bits or less, any simple mesh (simply connected, orientable, manifold with boundary) having t triangles and b "exterior" edges in its bounding loop. Edgebreaker can also encode fully general, non-manifold meshes with holes and handles, using $\log(h)+\log(t)+k$ additional bits per handle or hole in the mesh, where h is the number of handles or holes, and k is a small constant. (The symbol "log" stands for a base-2 logarithm throughout this paper.) For meshes with relatively few handles and few bounding edges, the compressed data requires between 1.5 and 2 bits of storage per triangle. This ratio may be even lower for compressing patches with a complex bounding loops and relatively few interior vertices.

Good compression ratios may be achieved by previously reported approaches for highly complex and uniformly tessellated meshes [Taubin98]. Edgebreaker leads to much simpler and more efficient coding and decoding algorithms. Furthermore, it significantly reduces the worst case output size of previously reported approaches and thus is particularly suitable for compressing large collections of simple disconnected meshes. Edgebreaker may be easily combined with lossy techniques that compress vertex location data. These techniques may for instance use a variable length coding of vertex corrective terms that are added to geometric estimates computed from previously received vertices [Taubin98]. Such estimates are based on incidence information, which therefore must be decoded before the vertices. Edgebreaker may be used to efficiently transfer the entire surface bounding a 3D model or the triangulation of a surface patch for which the bounding loops are already known. Because of its simplicity and versatility, Edgebreaker promises to play an important role in the emerging 3D data exchange standards for interactive graphic applications.

A.2. Outline of the paper

First, in the *Motivation* sub-section, we explain why compression of 3D model is important and why we chose to focus on triangle meshes. Then, in the *Background* sub-section, after introducing the appropriate concepts, terminology, and notation, we explain why it is important to compress the incident graph independently of the vertex location. Subsequently, in the *Simple Meshes* section, we present the Edgebreaker approach for simple meshes, demonstrate its correctness, and justify our claims on the compression ratios. We then discuss the merits of this approach in the context of *Prior Art*. Finally, we provide *Implementation* details and then explain how to support more *General Triangle Meshes*.

A.3. Motivation

A.3.a. The importance of compressing 3D models

Interactive 3D graphics already plays an important role in manufacturing, architecture, petroleum, entertainment, training, engineering analysis and simulation, medicine, and science. It promises to revolutionize electronic commerce and many aspects of human-computer interaction. For many of these applications, 3D data sets are increasingly accessed through the Internet. The number and complexity of these 3D models is growing rapidly, due to improved design and model acquisition tools, to the wide spread acceptance of this technology, and to the need for higher accuracy. In many of these applications, human productivity or satisfaction would be significantly enhanced by the possibility of an immediate access to remotely located 3D data sets for visual inspection or manipulation. Even when image-based rendering [Mark97, Mann97, Darsa97] and progressive transmission techniques [Hoppe96, Hoppe97] for adaptive resolution graphics are used to reduce the fraction of the 3D representation that must be transferred at any given time, geometry transfer remains the bottle-neck. The anticipated phone and network bandwidth increases will not, by themselves, suffice to offset the explosion of the complexity and popularity of 3D models. Consequently, it is urgent to develop optimal bit-efficient formats and associated compression and fast decompression algorithms for 3D models.

A.3.b. The need to focus on triangle-meshes

Although many representations have been proposed for 3D models [Rossignac94] polyhedra (or more precisely triangular meshes) are the de facto standard for exchanging and viewing 3D data sets. This trend is reinforced by the wide spread of 3D graphic libraries (OpenGL [Neider93], VRML [Carey97]) and other 3D data exchange file formats, and of 3D adapters for personal computers that have been optimized for triangles. Graphic subsystems can convert polygons and curved surfaces into an equivalent (or approximating) set of non-overlapping triangles, which may be rendered efficiently using hardware-assisted rasterizers [Rockwood89, Neider93]. But to avoid the cost of this runtime conversion, most applications precompute and store the triangle meshes. Therefore, triangle count is a suitable measure of a model's complexity and an appropriate target for current compression efforts [Rossignac97].

A.4. Background

In this section, we clarify our terminology and notation, and introduce the concepts and background results upon which our formulation and proofs are based.

A.4.a. Sets of triangles

We use the following notation. |X| denotes the number of elements in the set X. T denotes a set of topologically closed triangles T_i, for integer i in [1..|T|]. $\{T_i\}$ is the closed pointset of T_i and {T} is the union of these pointsets for all triangles in T.

A.4.b. Incidence graph and vertex list

A triangle is said to be *incident* upon its three vertices. Given a set V of vertices in 3D, a set T of triangles bounded by the elements of V may be entirely specified by an triangle/vertex *incidence graph*, denoted $G_T(V)$, which for each triangle identifies three of the vertices in V. For simplicity, and without loss of generality, we assume that the elements of V may be uniquely identified by integer numbers between 1 and |V|. This assumption, makes $G_T(V)$ (from now on simply denoted G) independent of V, except for the fact that G cannot reference more than |V| vertices. Note however that, V and G often obey strong constraints associated with specific topological restrictions on the domain, as the ones discussed below.

A.4.c. Manifold meshes with boundary

T is a *manifold mesh* if the relative interiors of its triangles are pairwise disjoint and if {T} is a connected two-manifold surface with boundary. This definition implies that the neighborhood of each point of {T} is homeomorphic to an open disk or to a half-disk [Massey67] and that any pairs of point in {T} may be joined by a curve that lies in the relative interior i{T} of {T}. (The relative interior of a surface S is the set of points of S whose neighborhood in S is homeomorphic to an open disk.) In a manifold mesh, each edge is bounding one or two triangles. Edges that bound two triangles are called *interior* edges. Edges that bound exactly one triangle are called *exterior* edges. Let b{T} denote the *relative boundary* of {T}, i.e., the set of points whose neighborhood in {T} is homeomorphic to a half-disk. b{T} is the union of all the exterior edges of T. The connected components of b{T} form one-manifold polygonal closed curves, called *loops*. Let V_I, be the subset of the vertices of V with a neighborhood in {T} that is homeomorphic to an open disk (these are the *interior* vertices of T). The others are called the *exterior* vertices of T and their set is denoted V_E.

A.4.d. Adjacency graph

Two triangles of T are said to be *adjacent* to one another if they are bounded by the same edge. The *adjacency graph* A(T), which captures all the adjacency relations between pairs of triangles in T, may be derived automatically from V and G. Its nodes correspond to the triangles and its links to the interior edges of T. In a manifold mesh, a triangle has at most three adjacent neighbors (at most one adjacent triangle per edge). The adjacency graph is often used to accelerate algorithms that compute local properties of the mesh and hence must traverse it in some systematic way from one triangle to an adjacent one. Numerous data-structures have been proposed that combine adjacency and incidence information (see [Rossignac94] for a review).

A.4.e. Simply connected meshes

We say that a mesh whose boundary is either empty or forms a single loop has *no holes*. If any closed one-manifold loop of edges of a mesh T separates the triangles of T into two disjoint sets, T is said to have *no handles*. For example, a torus has one handle. A connected manifold mesh with no holes and no handles is said to be *simply connected*.

A.4.f. Orientable meshes

Two opposite orientations may be associated with each triangle. A particular orientation may be implicitly defined by choosing an orientation for the normal associated with the triangle (the unit vector orthogonal to the plane that contains the triangle) or equivalently, by listing, in a clockwise order around that normal, the three vertex references that G associates with that triangle. This choice induces an edge orientation for the three edges in the triangle's boundary. A connected manifold mesh T is *orientable*, if there exists a consistent choice of triangle orientations so that the orientations induced upon each interior edge by the two incident triangles are opposite. Non-orientable meshes may be detected by selecting an arbitrary orientation for one triangle and by propagating it across interior edges to all triangles. The presence of interior edges with incident triangles that have conflicting orientations indicates that the mesh in non-orientable. Cutting through these edges (replacing each one of them by a pair of coincident exterior edges) transforms the surface into an orientable one, although the mesh may no longer be simply connected or manifold and may have an imbedding that is inconsistent with its topological representation. Throughout the rest of this paper, we will assume that the mesh is (topologically) orientable.

A.4.g. Simple meshes

We declare that T is a *simple mesh,* if and only if T is a manifold, orientable, and simply connected mesh. The following relation holds for simple meshes: $|T|=2|V_I|+|V_E|-2$. It is derived from the Euler equation for simply connected, planar 2-complexes with manifold boundary: $|T|-|E|+|V|=1$, where $|V|=|V_I|+|V_E|$ and where the number of edges $|E|$ accounts for the external and internal edges. Since there are $|V_E|$ external edges and $3|T|/2-|V_E|/2$ internal edges, we obtain $|T|-|V_E|-3|T|/2+|V_E|/2+|V_I|+|V_E|=1$.

When $|V_E|<<|V_I|$, there are approximately twice more triangles than vertices.

A.4.h. Representing a simple mesh

A simple mesh T may be defined by the set V of its vertices and by its triangle/vertex incidence graph, G. For example, V may be represented, using 12 bytes per vertex, as an array of vertices, each composed of three coordinates that are stored in floating point format. G may be represented using 12 bytes per triangle as an array of triangle descriptors, each composed of three integer indices identifying the entries of V. Many other equivalent representations may be easily and efficiently derived from this one.

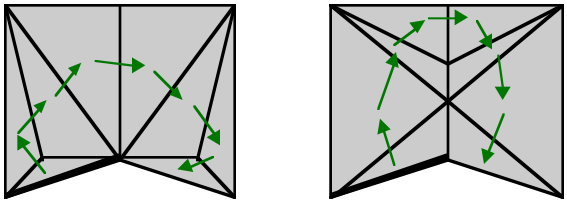
A.4.i. Triangle spanning tree

A spanning tree of the adjacency graph of a triangle mesh T is a binary tree whose nodes correspond to all the triangles of T and whose edges correspond to some of the interior edges of T. A depth-first traversal of such a spanning tree corresponds to a walk on the entire mesh that starts at the root triangle and recursively visits zero, one, or two of the neighboring triangles that have not been previously visited. Each triangle is visited by coming to it from an adjacent, previously visited triangle. (Our approach is based on such a traversal process and the term "Edgebreaker" stems from the metaphor of "breaking" an edge to penetrate onto an adjacent triangle.)

A.4.j. The triangle spanning tree does not encode G

The spanning tree may be encoded using 2 bits per triangle as follows. Each triangle is reached from a neighboring triangle through a "broken" edge. If the surface is orientable, the other two edges may be uniquely labeled as the *left* and the *right* edge. Each one of these may—but need not— correspond to an edge in the spanning tree of triangles. We thus need one bit for each one of these edges to indicate whether they are to be broken or not during the traversal.

If we could always derive a complete representation of G from the spanning tree of the triangles of T, we would have attained our objectives and would have a very simple scheme for encoding simple meshes using 2 bits per triangle. Unfortunately, the triangle spanning tree does not capture the entire topology of the incidence graph. For instance, the two meshes below have the same boundary and the same spanning tree (described by the following sequence of bit-pairs, one per triangle: 10, 01, 10, 01, 01, 10, 01), with the first broken edge identified by a thicker line. Therefore, a more elaborate scheme for encoding G is necessary.



A.4.k. Accuracy of vertex locations

Typically, tessellated models are used for visualization, interference detection, or finite-element analysis. They are often approximations of curved shapes, which may have to be represented with higher degree surfaces for manufacturing and more advanced simulation and analysis applications. Even when a model represents a shape that is polyhedral by nature, the accuracy of the model is often limited during its creation by numeric round-off errors in the computation of geometric intersections, by the limited resolution of input techniques during design, or by measurement errors. Applications for which such numeric inaccuracies or such crude polyhedral approximations of curved shapes are acceptable do not in general require that vertex coordinates be stored with full floating point precision, as long as the geometry preserves the important topological and adjacency relations.

Following [Deering95, Taubin98], we suggest to represent the vertex coordinates with k bits each, as integers between 0 and 2^k, defined over the smallest axis-aligned box that contains the model. For example, 10-bit quantization (k=10) will result in better than 0.5mm accuracy for any part of a car engine.

A.4.l. Vertex compression techniques

The storage for V may be further reduced by using a variable length encoding. Previously reported variable length coding schemes for vertices [Deering95, Hoppe96, Taubin98] are based on the following argument. If the compression and decompression algorithms may produce the same estimate for the location of each vertex, it suffices to transmit corrective displacement vectors. (The decompression algorithm will estimate the location of the next vertex and simply add it to the corrective vector.)

If the vertex coordinates are quantized to a small number of bits and if the estimates are good, many of the coordinates of the corrective vectors will be small integers. Entropy coding or other variable length schemes replace the frequently occurring integers with shorter codes.

Thus, in highly tessellated models with quantized coordinates, compression ratios for V depend primarily on the precision of the vertex estimates. For example, Taubin and Rossignac [Taubin98] have used vertex estimators based on a few ancestors in a spanning tree of vertices (where the edges of the spanning tree correspond to some of the edges of the mesh). For highly complex models with finely tessellated surfaces, their technique approaches 12 bits per vertex, which represents an average of only 4 bits per coordinate (or 6 bits per triangle). We anticipate that further compression could be obtained by exploiting more information about the topology of the incidence graph around each vertex and about the location of its neighbors.

A.4.m. G should be compressed independently of V

Simple meshes without boundary or with a small bounding loop have roughly twice more triangles than vertices. Therefore, although it may surprise some readers, 2/3 of the total storage in the simple representation outlined above for simple meshes is allocated to G. It is thus important to develop bit-efficient representations for G and to provide efficient compression and decompression algorithms.

Techniques for compressing G down to a few bits per triangle are already available [Turan84, Taubin98] and although further progress is needed, it cannot come at the expense of the compression ratios for V, which would then become the dominant cost factor. For example, Denny and Sohler [Denny97] have used a permutation of the vertices of V with respect to their lexicographic order to implicitly encode G for sufficiently large triangulations in the plane. One may envision extending their approach to the case of three-dimensional meshes. Unfortunately, techniques based on vertex permutations seem incompatible with the vertex compression techniques described above. Consequently, we need to focus on compression schemes that not only reduce the storage necessary for G, but, that at the same time allow its decompression before the decoding of V. This way, the location of each vertex in V may be estimated from those of its neighbors (identified through G) that have been previously decoded.

A.4.n. Edgebreaker compresses G independently of V

To meet the requirements outlined above, Edgebreaker compresses the incidence graph G and, in doing so, generates an ordering of the vertices of V. For simple meshes, as proven in the next section, the encoding O of G requires $2|T|+|V_E|-2$ bits and also permits to compute $|V_I|$ and $|V_E|$. Our decompression algorithm parses O; computes $|V_I|$, $|V_E|$; creates an empty array of vertices for V; and reconstructs G, which contains references to this array.

Then, if desired, it reads some encoding of V_E, decompresses them and fills in the first $|V_E|$ entries in the array. This latter step may be unnecessary, if a representation of V_E is already available to the decompression algorithm.

Finally the decompression algorithm reads an encoding of V_I, decompresses it, and fills the rest of the array. The coding scheme for the vertices of V_I may take advantage of the available description of their topological relation to other vertices in G, as explained above.

B. Simple meshes

In order to introduce a succinct and intuitive overview of the Edgebreaker approach, we focus in this section on simple meshes. Later in this paper, we explain how to generalize our scheme to non-manifold triangulated surfaces with an arbitrary number of handles and bounding loops and provide implementation details.

B.1. Compression process

Edgebreaker operates on a simple mesh. However, during the compression process, that mesh may be split into several simple meshes (interior-connected components that may share isolated vertices). They will be compressed one at a time. The current component will be denoted T. It has a single bounding loop, denoted B. Pointers that access the other components are stored in a stack.

The Edgebreaker compression is based on a sequence of primitive operations that each remove one triangle from T. At each stage of this process, an edge of the bounding loop is identified as the *active gate*. The active gate is stored at the top of the *stack of gates*. Initially, the edge connecting the first and the second vertex of B is the only gate on the stack. As T is split into connected components, the gates of the stack identify an edge in the bounding loop of each one of the connected components that have not yet been compressed. Through the rest of this paper, when there is no ambiguity, the term *gate* will be used to refer to the active gate.

The compression algorithm constructs two strings: the sequence of op-codes O and the sequence of vertex identifiers P. Initially these are empty, although in some applications, P may be initialized with the identifiers of the vertices of the original bounding loop, starting with the gate vertices. For each operation, the compression algorithm performs the following steps:

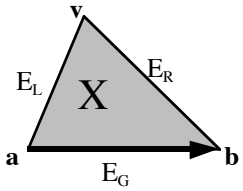
1. Identify X, the triangle of T that is incident upon the gate
2. Append to the compressed stream, O, a binary op-code that identifies the relation between the boundary of X and B
3. Remove X from T and update T and B (possibly splitting B into two loops)
4. Update the gate stack to identify the active gate and to keep track of loops to be processed later
5. If a new vertex has been inserted into B, mark it, and append its reference to the list of vertex identifiers P

P may be used to produce a sequence of corrective vectors that will be added by the decompression algorithm to its vertex estimates.

B.1.a. Edgebreaker moves

In this subsection, we describe the five topological situation that the Edgebreaker compression may find and explain how they are processed and encoded.

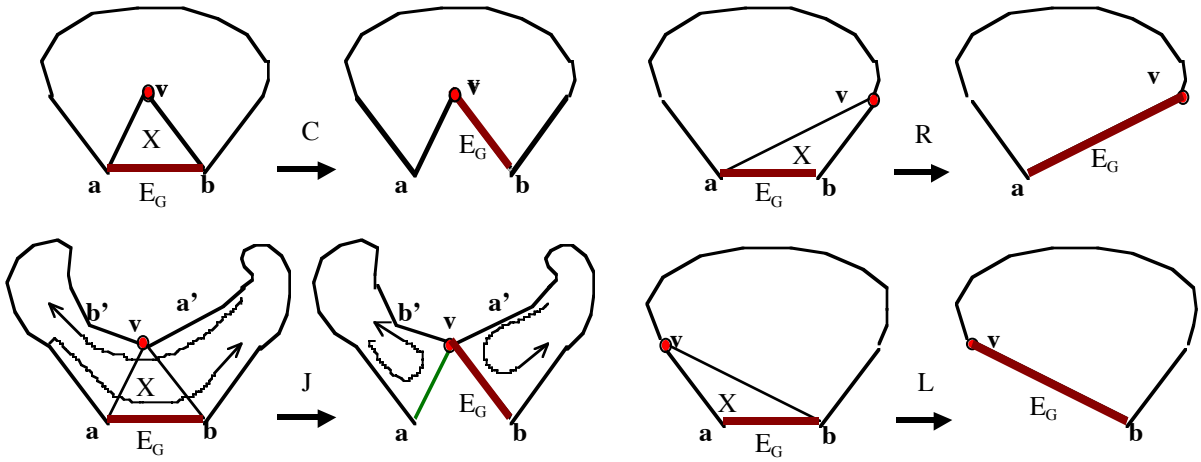
We use the following notation. E_G denotes the gate. Its orientation is induced from the orientation of B. Points **a** and **b** are the starting and the ending vertices of E_G. X is the only triangle of S that is incident upon E_G. **v** is the third vertex of X, i.e., the vertex not bounding E_G. E_L is the edge from **v** to **a** and E_R is the edge from **b** to **v**. (See figure below).



We need only to distinguish a few different types of situations among those encountered by the compression process. They are differentiated by the inclusion status of E_L, E_R, and **v** with respect to B. For each one of the 8 combinations of these classifications, the table below provides the name and a possible binary code of the associated operation, its effect on the number of edges (edge-count) in the boundary of the remaining mesh (after the triangle X is removed), and the associated actions. Note that, because ($E_L \subset B$ **OR** $E_R \subset B$) $\Rightarrow \mathbf{v} \in B$, three of the combinations are impossible. We can exploit this observation to reduce the bit-count of some op-codes. For example, we can use a single-bit op-code for C, which in general is the most common operation.

v ∈ B	$E_L \subset B$	$E_R \subset B$	op-code	operation	edge count	execution
0	0	0	0	C	+ 1	E_G=vb
1	0	0	100	J	+ 1	Split B at v into two loops. E_G= av. Push stack. E_G=va'.
0	0	1	-	impossible		
1	0	1	101	R	– 1	E_G=av
0	1	0	-	impossible		
1	1	0	110	L	– 1	E_G=vb
0	1	1	-	impossible		
1	1	1	111	F	– 3	If not_empty(stack) then pop stack.

The pre-conditions and post-conditions of the C, J, R, and L operations are illustrated in the figure below.

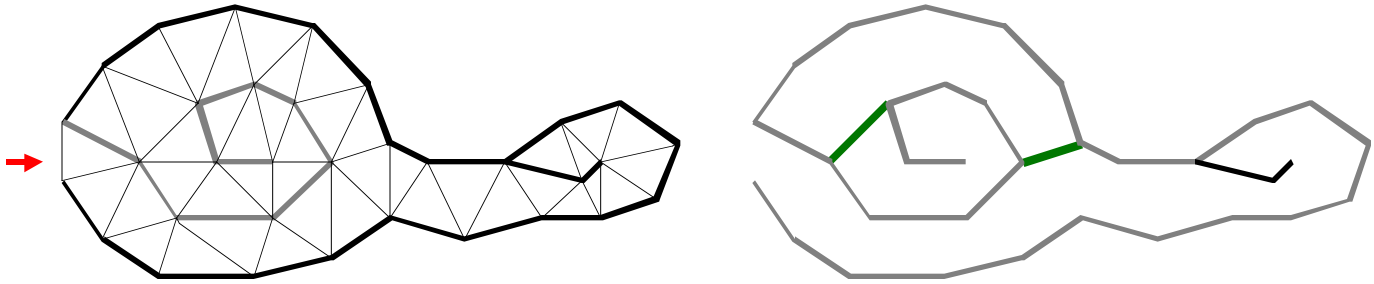


Note that the J operation, identified by the situation where $E_L \not\subset B$ **AND** $E_R \not\subset B$ **AND** $v \in B$, corresponds to a change of topology in {T}, which separates i{T} into two connected components. Each component is bounded by a separate loop. Both loops intersect only at **v**. The edge **vb** is placed at the top of the stack and becomes the active gate. The edge **av** is placed immediately below it in the stack. It will be processed later, once the connected component of i{T} that is incident upon edge **vb** is encoded.

The F operation, not shown above, corresponds to the trivial case where $E_L \subset B$ **AND** $E_R \subset B$ **AND** $v \in B$. It implies that X is the only triangle remaining in T. In such cases, the Edgebreaker compression simply appends the op-code F to O, pops the stack, and, if the stack is not empty, proceeds to "break" the next gate at the top of the stack.

B.1.b. Corridors

The compression process breaks all edges that have been gates. These edges are marked with thinner lines in the figure below (left). The arrow on the left indicates the initial gate. The broken edges have been removed (right) to expose corridors that show the path of the Edgebreaker compression and decompression. The thick edges on the right mark gates that have been stored below the top of the stack as inactive gates. They mark the entrances to new corridors that branch off previously processed ones. The corresponding sequence is O=(CRCRRCRCRJLRLRCRCRRRRLFRCRCRCRRJFCRRCRRRLF).



B.1.c. Compressed format

If the original boundary B has been previously decoded and the original gate is identified by some simple convention, the compressed data stream contains:

1. The sequence O of op-codes produced by the compression algorithm
2. Some encoding of the vertices of V_I in the order specified in the vertex list P

These two strings may be further compressed by general purpose compression schemes, not discussed here.

If the original boundary is not available prior to decompression, the compressed data stream contains:

1. The sequence O of op-codes produced by the compression algorithm
2. Some encoding of the vertices of V_E in their order along B starting with the vertices of the original gate, followed by some encoding of the vertices of V_I in the order specified by vertex list P

Note that the vertices of V_E and V_I may be encoded in any desired way, and in fact may be sent in a different order, provided that a convention enables the decompression algorithm to recover the order specified by P.

B.2. Storage cost for the incidence graph

When using the binary op-codes suggested in the table above, we need 1 bit to encode each C operation and 3 bits to encode each other operation. Hence, denoting by |C| the number of C operations in O and, using a similar notation for the other operations, we can express the total number of bits needed to encode O as b= |C|+3(|J|+|L|+|R|+F|).

Because there is a one-to-one association between the vertices of V_I and the triangles processed by a C operation, we have $|C|=|V_I|$. Consequently, $|J|+|L|+|R|+|F|=|T|-|C|=|T|-|V_I|$. Thus $b=|V_I|+3(|T|-|V_I|)=2|T|+(|T|-2|V_I|)$. Given that, as mentioned in the "Background" sub- section, $|T|-2|V_I|=|V_E|-2$, we obtain $b=2|T|+|V_E|-2$.

B.2.a. Simple meshes without boundary

To encode a *simple mesh without boundary*, such as the entire surface that bounds a manifold 3D solid, it suffices to "cut open" one of its edges, declare it to be the initial loop, B, and include the encoding of its two vertices at the top of the vertex list. In that case, $|V_E|=2$ and $b=2|T|$, which is exactly 2 bits per triangle.

B.2.b. Simple meshes with relatively simple boundary

For simple meshes with boundaries, for which $|V_E|<<|V_I|$, we have $|V_E|<<|T|$, and thus $b\approx 2|T|$.

B.2.c. Impossible sequences

The CL and CF sequences of operations correspond to situations where two triangles are identical (have the same vertices). By definition, these situations are impossible in simple meshes. We can exploit this constraint to increase the expected compression ratio of Edgebreaker by using a slightly more complex coding scheme. We use two different code sets:

- the general code set proposed earlier for operations that do not follow a C operation
- a special code set for operations that follow a C

The special code set is still 0 for C, but may be now reduced to a 2-bit op-code for the other two operations: 10 for J, and 11 for R. In the worst case, with long sequences of consecutive C's, this scheme has no effect on the bit-count. At best however, when all C's are separated, it reduces the bit-count to an average of 1.5 bit per triangle (because there are as many C's as other operations).

B.2.d. Simple meshes with relatively complex boundaries

When Edgebreaker is used to compress small surface patches with a relatively large number of edges in their boundary, the above binary codes will not exceed an average of 3 bits per triangle, but are not optimal. Because, in such cases, the R operation is the most frequent in the sequence, the op-codes proposed earlier should be replaced by others, where R is a one-bit code (say 0) and the other four operations have 3-bit codes. Under these new conditions, b=3|T|-2|R|, which implies that if most of the triangles correspond to an R operation (which is the case for a fan of triangles), the sequence representing G may be compressed down to 1 bit per triangle. For example, using this scheme for the sequence O=(CRCRRCRCRJLRLRCRCRRRRLFRCRCRCRRJFCRRCRRRLF) of the above figure require an average of 1.48 bits per triangle.

B.2.e. Customized codes

For meshes that do not fall in these two categories (boundary-heavy or interior-heavy), we suggest a post-processing compression step, which would compute the optimal op-code assignment for each operation, taking into account their frequencies and the constraints on impossible sequences. The resulting codes would be transmitted before O, using some convention. For example, we may agree to encode the bit length (coded using 2 bits) followed by the actual code for: C following a C, R following a C, F following a C, and then code-descriptions for the occurrences of C, R, L, J, and F that do not follow a C. This table will take at most 42 bits and thus this customization technique may not be appropriate for encoding small meshes, for which it may suffice to use a one-bit switch to select between the code for the interior-heavy or for the boundary-heavy meshes. Progressive coding schemes [Ziv77, Welsh84, Neslon89] may also be used.

B.3. Justification of completeness and correctness

In this subsection, we show that the process terminates and that sequence of op-codes O produced during the compression process defines completely the topology of the mesh, i.e., the incidence graph G of T. The table above shows that the preconditions for the L, R, C, J, and F operations are mutually exclusive and cover all possible cases. They all decrement the triangle count in T, which implies that the compression process terminates. The decompression algorithm executes the sequence encoded in O and creates the triangles instead of removing them, but essentially follows the same sequence of moves, and thus terminates as well.

We need to prove that the decompression algorithm can: (1) compute $|V_E|$, (2) compute $|V_I|$, and (3) identify the vertices for each triangle. We describe below how this information may be derived from O.

B.3.a. Computing the number of interior and bounding vertices

Since only C operations require the introduction of new vertices, $|V_I|=|C|$.

We know that at the end of the whole process, the boundary of {T} must have zero edges. If we can extract from O how many edges have been added or deleted by the compression process, we will know the initial length of B. The "edge count" column in the above table indicates how each operation affects the total count of edges (and thus of vertices) in B. For example, R deletes two edges from B, but exposes a new one, thus decreases the edge count by 1. We can track the total count regardless of the topological changes in the

boundary that may be produced by J operations. These edge counts increments lead to the following formula: The number *b* of edges (and thus of vertices) in the initial loop, B, is b=3|F|+|L|+|R|−|C|−|J|.

Consequently, by counting the number of C operations we can compute $|V_I|$ and by computing b according to the above formula, we can obtain $|V_E|$.

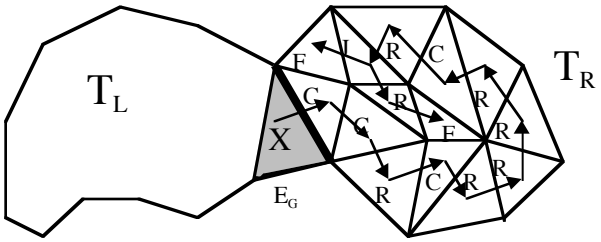
B.3.b. Locating the third vertex

Given that the gate identifies two of the vertices of X, the decompression algorithm needs only to derive from O the identification of the third vertex, **v**.

- During a C operation, **v** is simply the next vertex in the vertex list specified in P.
- During an R operation, **v** may be easily identified as the vertex that follows the gate in the circular ordering of B.
- Similarly, during a L operation, **v** precedes the gate in B.
- During the F operation, **v** is the only vertex in B that is not in the gate. It follows and precedes the gate.
- During a J operation, we know that **v** lies somewhere along B, but we need to compute its *offset* from the gate. The *offset* is the number of vertices that must be skipped before reaching **v,** when we start from the end of the gate and walk along B according to its orientation. We explain below how the offset calculation is performed.

During a J operation, the compression process removes a triangle X and splits T into simple meshes T_L and T_R. The triangles of T_R form $\{T_R\}$, the interior-connected component of i{T} that is incident upon E_R. The compression process must remove all the triangles of T_R before popping E_L to the top of the stack and proceeding to compress T_L. Thus the offset equals the number of edges in the boundary of T_R minus 2. (E_R does not count and there are one more edges than skipped vertices.) We can calculate the number of edges in the bounding loop of T_R using the formula for *b* developed above. However, we must be able to identify which subset of O corresponds to the op-codes for compressing T_R. That subset starts right after the current J operation and lasts until we encounter the corresponding F, that is the first F that does not cancel a previous J in that string. (We simply initiate a counter to 1, increment it for each J, decrement for each F, and stop when the counter becomes zero.) The offset *o* in that sub-sequence is identified by *o*=3|F|+|L|+|R|−|C|−|J|−2.

For example, the compression process illustrated in the figure below, would produce for T_R the string CCRCRRRRCRJRFF, for which *o*=3x2+1+6−4−1−2=6.



B.4. Decompression process

The decompression algorithm first computes the number, $|V_E|$, of edges in the initial bounding loop and the offsets for all the J operations, using the formulae developed in the previous subsection. This is achieved through a single traversal of the sequence O of op-codes and hence has linear complexity. Then the decompression algorithm reconstructs the triangle/vertex incidence for each triangle by maintaining a set of bounding loops (circular linked lists) referenced by the gates of the stack. At each step, the active gate identifies two vertices of the current triangle. The corresponding op-code (combined with the precomputed offsets for J operations) identifies the third vertex and indicates how the loops must be updated and where the next gate is. The decompression algorithm is capable of detecting the end of the O string as the first F that does not cancel a previously encountered J.

B.4.a. Decompression performance improvements

The initial traversal of the entire string O may be avoided by encoding explicitly: $|V_E|$ (if it is not already known by the decompression algorithm), $|V_I|$, and the offset corresponding to each J operation. The offset may follow the corresponding J op-code in O and may be represented by a *k*-bit integer, where *k* is the smallest integer for which 2^k equals or exceeds the number of edges in the boundary of the current mesh.

If the vertex encoding are interleaved with the O stream, the description of each vertex may follow the corresponding C operation. With this convention, the mesh may be decoded in-line and the corresponding triangles may be generated one at a time, without the need to look ahead. Thus, very complex meshes could be decoded and rendered one triangle at a time without having to store the previously created triangles nor the previously used vertices, except for those in the stacked bounding loops.

C. Prior art

We organize prior art into five categories: uncompressed data structures, triangle strips, vertex insertion, graph encoding, and vertex permutations. We provide here a brief overview of these approaches and compare their expected or worst case compression capabilities.

C.1. Uncompressed data structures

C.1.a. Vertex table

As suggested earlier, V and G may be stored separately and transmitted independently. V may be represented as an ordered sequence of vertices, each described by its three coordinates. G may be represented by an unordered set of triangle descriptors, each composed of three integer indices to the vertex array. This model requires $\lceil 3\log(|V_I|) \rceil$ bits per triangle for G (where $\lceil x \rceil$ denotes the lowest integer greater than x).

C.1.b. Independent triangles

Storing each triangle independently of all other triangles as the list of floating point representation of the coordinates of its three vertices would require 36 (=3x3x4) bytes per triangle. In such a simple representation, the incidence is coded implicitly by the order of the vertices and does not require any storage. Instead, vertices in V are repeated on average 6 times.

If the vertex coordinates were stored as integers of 10 bits each (as argued for in the background section), this representation would require 90 bits per triangle.

To avoid storing the 30 bits of each vertex more than once, we can envision using 1 bit per vertex reference to indicate if the next entity in the input stream of vertices is a 30-bit location of a new vertex or if it is a reference to a previously decoded vertex location. If the number of previously decoded vertices is n, then the reference requires $\log(n)$ bits. Since n is known both to the compression and decompression algorithms, they can use the optimal number of bits for encoding each reference.

Although the count n increases from 1 to $|V_I|$, it is possible that all references be invoked only after all vertices have been decoded. Consequently, in the worst case, this representation would require $31|V_I|+5|V_I|\log(|V_I|)$ bits of storage. If we subtract $30|V_I|$ for the vertex coordinates, the cost of encoding G is $|V_I|(5\log(|V_I|)+1)$ bits, or equivalently $2.5\log(|V_I|)+0.5$ bits per triangle, if we assume twice more triangles than vertices and ignore the impact of V_E.

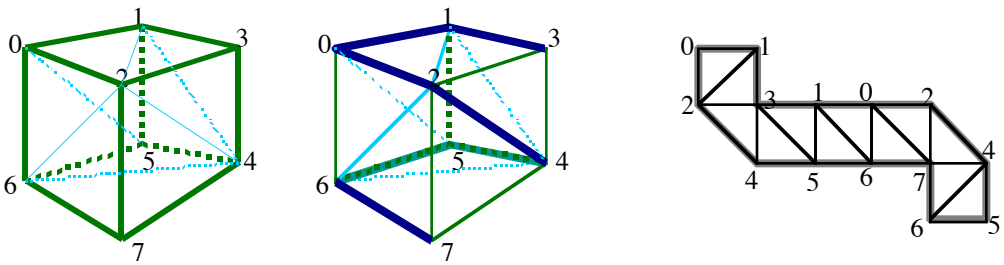
C.1.c. Bar-Yehuda

The $\log(|V_I|)$ factor may be reduced by sorting triangles so as to reduce the total number of vertices that may be referenced at any time [Bar-Yehuda96] to $13(|V_I|)^{0.5}$. With this approach, the above technique requires only $1.25\log(|V_I|)+0.5+2.5\log(13)$, or roughly $1.25\log(|V_I|)+9.25$, bits per triangle.

C.2. Strips

C.2.a. Triangle strips

A representation based on triangle strips, supported by OpenGL [Neider93] and other graphic libraries, is used to reduce the number of times the same vertex is transferred and processed by the graphics subsystem. Basically, in a triangle strip, a triangle is formed by combining a new vertex description with the descriptions of the two previously sent vertices, which are temporarily stored in two buffers. Each new triangle, X, shares an edge with the previous triangle in the strip. Using a convention to orient the surface of the strip, we can label the other two "free" edges of X as the *left* and the *right* edge. One bit per triangle suffices to indicate whether the triangle is incident upon the left or the right edge of the previous triangle. The first two vertices are the overhead for each strip, so it is desirable to build long strips, but the automation of this task remains a challenging problem [Evans96]. Instead of using such a left/right bit, OpenGL requires to alternate between left and right edges throughout the strip. (Two consecutive right or left "moves" may be implemented without breaking the strip by encoding a vertex twice.) The figure below shows a triangulated cube and the corresponding triangle strip flattened out.



Note that in our example, each vertex, except for 3 and 7, is encoded twice. This is in general the case for triangle strips.

Let us assume that, as in the case of independent triangles, we avoid vertex replication and encode, in lieu of a replicated vertex, a reference to a previously decoded one. Assuming strips of length $k >> 1$ and one bit per triangle to indicate whether the next triangle is

attached to the left or the right edge of the current one, we need a total of $|T|+|V_I|\log(|V_I|)$ bits to represent G, which is equivalent to $1+0.5\log(|V_I|)$ bits per triangle, with the same simplifying assumptions as above of a simple mesh with a relatively complex interior.

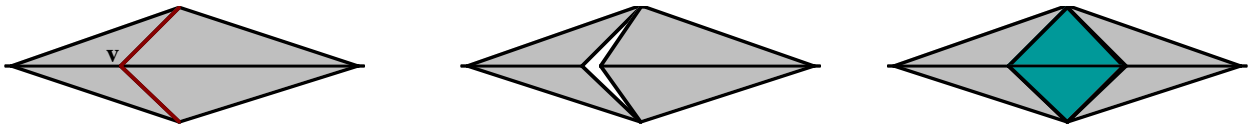
C.2.b. Deering

Deering's approach [Deering95] is a compromise between a standard triangle strip and a general scheme for referencing any previously decoded vertex. Deering uses a 16 registers cache to store temporarily 16 of the previously decodedvertices for subsequent uses. He suggests to use one bit per vertex to indicate whether each newly decoded vertex should be saved in the cache. Two bits per triangle are used to indicate how to form the next triangle. One bit per triangle indicates whether the next vertex should be read from the input stream or retrieved from the cache. 4 bits of address are used for randomly selecting a vertex from the stack-buffer, each time an old vertex is reused. Assuming at best that a ratio of 14/16 vertices are reused from the cache and that 2/16 vertices must be reused while they are not in the cache, the total cost of Deering's approach, when combined with a random access to previously decodedvertices would be $4.5+0.125\log(|V_I|)$. Note that it may prove difficult to build such generalized strips while maintaining a low count of vertex replication.

C.3. Vertex insertion

C.3.a. Hoppe

Hoppe's Progressive Meshes [Hoppe96] permit to transfer a 3D mesh progressively, starting from a coarse mesh and then inserting new vertices one by one. Instead of a vertex insertion to split a single triangle, as suggested in [Dobkin85] for convex polyhedra, Hoppe applies a vertex insertion that is the inverse of the edge collapse operation popular in mesh simplification techniques [Hoppe93, Ronfard96, Heckbert97]. A vertex insertion identifies a vertex **v** and two of its incident edges. It cuts the mesh open at these edges and fills the hole with two triangles (see figure below). **v** is thus split into two vertices.



Each vertex is transferred only once in Hoppe's scheme. The cost of G for each vertex is the identification of one of the previously transferred vertices (on average more than $0.5\log(|V_I|)$) plus the cost of identifying two of the incident edges (5 bits are sufficient if no vertex is bounding more than 32 edges). Thus, the cost per triangle would be more than $2.5+0.25\log(|V_I|)$.

C.4. Graph encoding

C.4.a. Turan

Turan has shown that the structure of a labeled planar graph may be encoded using slightly less than 6|T| bits [Turan84]. Having a constant number of bits per triangle has a significant advantage over the previous approaches, which all include a $\log(|V_I|)$ factor, especially for highly complex meshes. Turan builds a vertex spanning tree and uses it to represent the boundary of a topological polygon of 2|V|-2 edges. The structure of this tree is encoded using 4|V|–4 bits. There are at most 2|V|-5 edges that do not belong to the vertex spanning tree. These may be encoded using 4 bits each. The overall cost is thus, 12|V|-24 bits.

C.4.b. Taubin and Rossignac

The Topological Surgery method recently developed by Taubin and Rossignac [Taubin98] also builds a vertex spanning tree of T that splits the surface of the mesh into a binary tree of corridors (generalized triangle strips). They encode both trees using a run length code, which for highly complex meshes yields an average of less than two bits per triangle. In addition, they use one bit per triangle to indicate whether the next triangle in a corridor is attached to the left or the right edge of the previous one. The compactness of the encoding of both trees comes from the fact that, by construction, both trees tend to have very few nodes with more than one child. Sequences of consecutive nodes with a single child are grouped into runs and encoded by simply storing their length, using $\lceil |V| \rceil$ bits. For pathological cases, with a non-negligible proportion of multi-child nodes, their approach does no longer guarantee a linear storage cost.

The vertices are stored in the depth-first traversal order of the vertex spanning tree. The entire mesh is represented by the list of vertex coordinates, an encoding of the sparse vertex and corridor trees, and the string of left/right bits. The application of this technique for VRML files is discussed in [Taubin98b].

C.5. Vertex permutation

C.5.a. Denny and Sohler

Inspired by [Kirkpatrick83] and improving on [Naor90, Snoeyink97], Denny and Sohler have recently proposed a technique for encoding G for *planar* triangulations of sufficiently large size as a permutation of the vertices in V [Denny97]. They show that there are less than $2^{8.2|V|+O(\log|V|)}$ valid triangulations of a planar set of $|V|$ points, and that for sufficiently large |V|, each triangulation may be associated with a different permutations of these points (there are approximately $2^{|V|\ln(|V|)}$ such permutations). Their approach requires transmitting an auxiliary triangle that contains the entire set and the vertices of V in a suitable order, computed by the compression algorithm. The decoding process sorts V lexicographically and then sweeps over the progressively refined triangulation, from left to

right. At each vertex of V, the enclosing triangle is identified [Lee77] and the vertex is inserted according to the incidence relation derived from the permutation. The vertices of V are transmitted progressively in batches. The successive batches are constructed through repetitive plane-sweeps, during which all vertices with degree at most 6 are removed incrementally and the resulting holes re-triangulated. For each point, the information needed to reconstruct that triangulation is encoded in the permutation of the vertices of the batch. The batches are decompressed in inverse order. Although for sufficiently complex models the cost of storing G is null, the unstructured order in which the vertices are received and the absence of the incidence graph during their decompression makes it difficult to use predictive techniques for vertex encoding.

C.6. Summary of storage requirements

The table below compares the storage cost of the previously described techniques for a simple mesh with a negligible number of bounding vertices. The cost is expressed as the number of bits per triangle.

Vertex table	$3\log(\|V_l\|)$	
Independent triangles	$2.5\log(\|V_l\|)+0.5$	
Bar-Yehuda	$1.25\log(\|V_l\|)+9.25$	Require complex algorithms to compute sequence.
Triangle strips	$0.5\log(\|V_l\|)+1$	
Deering	$0.125\log(\|V_l\|)+4.5$	Assuming that we can build general strips with minimal vertex repetition.
Hoppe	$0.25\log(\|V_l\|)+2.5$	Assuming a constant bound on the number of edges per vertex
Turan	6	Works for general planar graphs
Taubin and Rossignac	1.5 - 3.5	Only when trees have long runs.
Denny and Sohler	0	Only for sufficiently complex 2D triangulations. Permutes vertices.

The Edgebreaker approach introduced here requires only 2 bits per triangle and is simpler than the previously proposed approaches of Deering, Bar-Yehuda, Taubin&Rossignac, and Denny&Sohler.

D. Implementation

We still focus in this section on simple meshes. The next section explains how to extend our approach to more general meshes. Although Edgebreaker may be implemented in a variety of ways, we introduce here a convenient data-structures and suggest an efficient and simple implementation of some of the details.

D.1. Input data structures

D.1.a. The Tripledge data-structure for simple meshes

The input data-structure should combine the *geometry* (vertex location) of the simple mesh and its *connectivity* (triangle/vertex incidence and triangle/triangle adjacency graphs). We represent geometry by an array W of vertices, each represented by its three coordinates. However, Edgebreaker never accesses the vertex coordinates, which hence may be represented in a variety of ways.

We propose a new, compact data-structure, called *Tripledge*, to represent the connectivity, which combines the information in G and A. Tripledge is a minor variation on several previously proposed datastructures for polyhedral models (see [Rossignac94] for a variety of references). It is based on two parallel arrays of integers: S and R, which we use to encode half-edges. A *half-edge*, denoted **h** identifies an edge/triangle incidence association. Thus, an edge of a simple mesh T may correspond to one or two half-edges. The half-edge inherits its orientation from the associated triangle.

Tripledge use the following convention. S[**h**] in an integer which defines in W the starting point of the oriented half-edge **h**. R[**h**] is an integer which identifies the reverse half-edge with the same endpoints, but with the opposite orientation. Thus R[R[**h**]]=**h**. W[S[**h**]] is the starting vertex of the half-edge **h** and W[S[R[**h**]]] is its end-vertex.

Instead of storing references from each half-edge to the other two half-edges of the same triangle, Tripledge define these associations implicitly, by ordering half-edges in S and R so that (W[S[**t**]], W[S[**t**+1]], W[S[**t**+2]]) are the three vertices of triangle **t** listed in counterclockwise order. This variation reduces storage and facilitates referencing individual half-edges.

D.1.b. Building the Tripledge data structure from G

The S and R tables of Tripledge may be efficiently constructed as follows. First traverse the triangles and fill in the S array. Then, create an auxiliary table containing, within each record, the integers identifying the two vertices of each half-edge and the integer identifying the corresponding half-edge. Make sure that the vertex references in each record are sorted in lexicographic order. Sort the array using the identifiers of the two vertices as fields. Now, all pairs of opposite half-edges are consecutive in this array and the associated half-edge identifiers may be used to update R. Half-edges that do not have a partner with the same vertices correspond to the initial set of bounding edges.

D.1.c. Accessing the consecutive half-edges of a triangle

Let the function N(**h**) return 3(**h** DIV 3) + ((**h**+1) MOD 3), which is the next half-edge after **h** around the triangle associated with **h**. Let P(**h**) return N(N(**h**)), the previous half-edge in the triangle. The vertex opposite to **h** in that triangle is identified by the integer S[P(**h**)].

D.2. Output format

Compression produces two arrays: P and O. P is the list of integers identifying the entries in W as they are accessed by Edgebreaker. O is the sequence of binary op-codes for the traversal of the mesh.

D.3. Auxiliary data-structure

D.3.a. Representing the bounding loops

During compression and decompression, we need to represent the bounding loops of the simple meshes that need to be encoded. Each bounding loop is represented by a doubly-linked list of *edges*. Each such edge **e** contains an integer reference, **e.**h, which identifies the corresponding half-edge in R and S. Let **e.**p and **e.**n return respectively the previous and the next edge in the doubly linked list. A reference to one edge of each loop is stored in the stack. The top of the stack references the active gate.

D.3.b. Constructing the initial bounding loops

The initial bounding edges may be identified through the sorting procedure described above for the construction of R. To order them along the initial bounding loops, it suffices to construct an array of records representing each bounding edges twice (once by the pair of integers that identify its start and end vertices and once by the opposite pair which identifies the end vertex first. The records also indicate whether the order of the pair of integers correspond to the orientation of the associated half-edge or to the reverse. Finally, the records also contain a reference to the edge.

Sorting the array makes adjacent pairs of bounding half-edges consecutive. In fact each pair appears twice. The sequence of half-edges around the bounding loops may be easily constructed by updating the **e.**p and **e.**n references for each pair of consecutive entries in the sorted array.

D.3.c. Vertexmark

To speed up the distinction between C and J operations during compression, we also use a one-bit mark M[**v**] for each vertex **v** in W. In the next section, we will expand the role of this mark to serve as a loop identifier for meshes with handles.

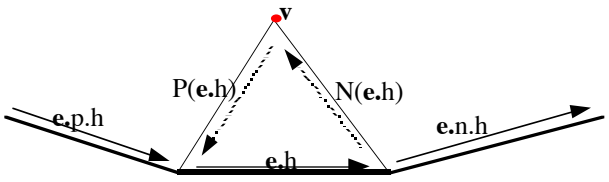
D.4. Compression details

Given the gate **e** (top of the stack), which identifies the corresponding half-edge **e.**h, we identify the opposite vertex **v** by S[P(**e.**h)]. Then we identify the appropriate operation and update the bounding loop as follows.

D.4.a. Operation selection

The following sequence of tests selects the appropriate operation. The associated entities are shown below.

```
IF NOT M[S[P(e.h)]]              ## If v is not marked
        THEN C
        ELSE IF e.p.h==P(e.h)
                THEN IF e.n.h==N(e.h) THEN F ELSE L
                ELSE IF e.n.h==N(e.h) THEN R ELSE J
```

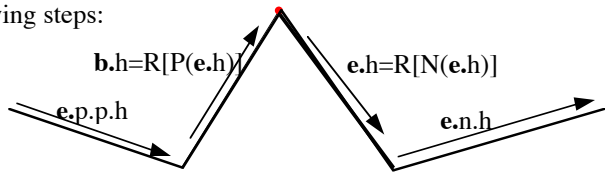


These tests and the corresponding operations are repeated until the stack is empty.

D.4.b. C operation

If **v** is not marked, we must execute a C operation, which requires the following steps:

- Append the op-code for C to O
- Append the identifier **v** to P
- Mark **v**
- Update the current boundary as follows
    1. Create a new edge **b**
    2. Insert it in the doubly linked list before **e**, for example through: **b.**p=**e.**p; **b.**n=**e**; **e.**p.n=**b**; **e.**p=**b**;
    3. Update references to half-edges: **b.**h=R[P(**e.**h)]; **e.**h=R[N(**e.**h)];



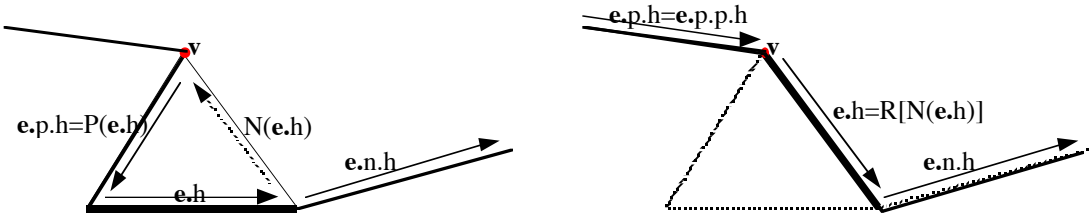
The result is shown on the right. The new gate is marked with a thicker line.

D.4.c. F operation

Edgebreaker deletes the elements of the current loop and appends the op-code for F to O. Then the compression algorithm pops the stack. If the stack is empty, the process terminates.

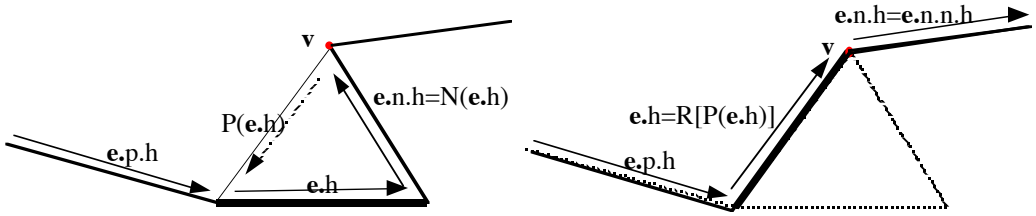
D.4.d. L operation

Edgebreaker deletes **e.**p from the list of bounding edges and changes the gate reference to the half-edge R[N(**e.**h)], as shown below. It appends the binary op-code of L to O.



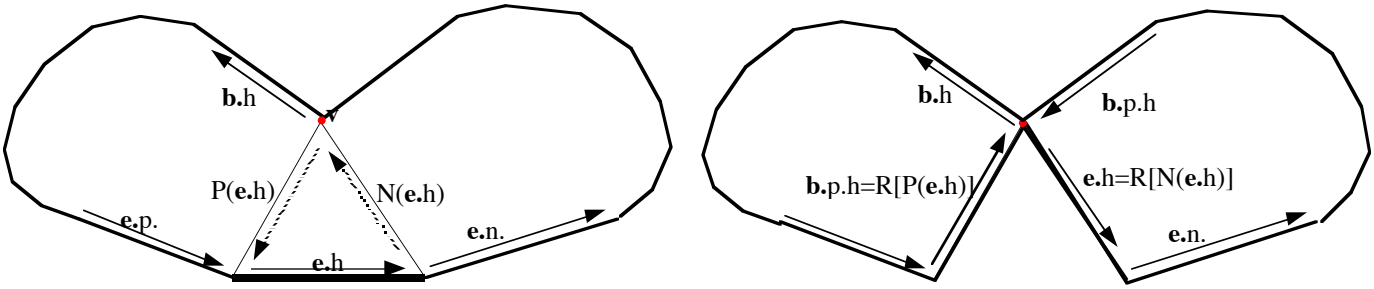
D.4.e. R operation

Edgebreaker deletes **e.**n from the list of bounding edges and makes the gate reference the half-edge R[P(**e.**h)], as shown below. It appends the binary op-code of R to O.



D.4.f. J operation

During a J operation, the compression algorithm must first traverse the bounding loop until it finds an edge **b**, such that S[**b.**h] equals **v**. Then, as shown below, it performs the same operation as for C followed by a split of the bounding loop at **v**. It replaces the gate with an edge that references R[P(**e.**h)] and then pushes the stack to load the new gate which references R[N(**e.**h)]



D.5. Decompression details

Decompression pre-computes the J-offsets as suggested earlier. Then, reading the operations in O, it performs the corresponding operations and produces the triangles one by one.

Note that decompression does not need to build the representation of the final mesh, nor maintain the auxiliary data-structure. It may for example output the triangles independently of each other or in triangle strips for graphics. Decompression still needs to maintain the set of bounding edge loops and the stack, but the bounding edges need not point to half-edges. They may instead simply represent the corresponding vertices or indices into a vertex array, if one is available.

J locates the third vertex by walking around the loop skipping vertices as indicated by the corresponding offset. C simply allocates the next available vertex id to the third vertex.

The logic of updating the bounding loop is the same as for compression. However, P() and N() functions must be replaced by the corresponding functions that operate on the newly created triangle.

E. General triangle meshes

As we pointed out earlier, Edgebreaker's traversal of the mesh during compression and decompression is a topological traversal, which does not depend on the location of the vertices. In fact, some choices of vertex locations may imbed the Tripledge representation in such a way that the corresponding surface self intersects at vertices, edges, or even in the relative interior of triangles.

E.1. Topological constraints on simple meshes

Having said that, we need to point out that not all meshes represented in the Tripledge format can be compressed by the Edgebreaker algorithms described earlier for simple meshes. Several restrictions apply:

1. Each vertex may only have one *cone* of incident triangles. (More precisely, the boundary of the star of each vertex must be a connected one-manifold curve, where the star is the union of the triangles incident upon the vertex)
2. The mesh must be orientable.
3. The mesh must not have any holes (i.e., its external edges form a connected set)
4. The mesh must not have any handles.

E.2. Splitting the mesh into simple pieces

An arbitrary mesh may always be converted into a collection of simple meshes by replicating vertices and introducing cuts that split the Moebious strips and make it orientable, that merge the holes with the outer boundary, and that cut through handles. Therefore, with the appropriate preprocessing, an arbitrary mesh may be converted into one or several simple meshes and the compression/decompression algorithms may be applied to the simple meshes independently. However, it may not be easy to locate the appropriate cuts and in general it is not desirable to replicate vertices as a preprocessing step to compression. We describe below a more economical approach.

E.3. Non-manifold meshes

Isolated non-manifold vertices which do not satisfy condition 1 listed above (a single cone of incident triangles) must be replicated so that the triangles of each cone refer to a different entry in W. This may be accomplished by considering their star and by identifying the connected components of the boundary of that star. The boundaries of non-manifold polyhedra are of particular interest, since they are often produced by solid modeling operations. They may be represented as *pseudo-manifolds* in our Tripledge data-structure proposed above and compressed as if they were manifold meshes with handles and boundaries. The simplest approach to this conversion is to identify all non-manifold edges and treat them as if they were boundaries of holes. More economical solutions that minimize the number of replicated vertices are under investigation and will be discussed in a separate paper. More general meshes, which include self-intersecting surfaces and dangling faces may also be converted to pseudo-manifolds by assigning triangle/triangle adjacency relations to pairs of triangles that share an edge.

In the remainder of this paper, we will assume that non-manifold vertices have been appropriately replicated. We also assume that the surface has been cut so that it is orientable.

E.4. Holes

To correctly compress simple meshes with holes, we precompute the bounding loops for all the holes (as explained earlier) and keep references to them in a separate array. The boundaries of the different holes may be unambiguously identified as the connected components of the relative boundary of {T}. We mark each vertex of the initial boundary with the integer identifier of its hole. The compression algorithm progresses as before, except for those J operations for which the third vertex **v** lies on the boundary of a hole that has not yet been merged with the current outer loop. In that case, we merge the two loops into one (instead of splitting the current loop). We thus need to be able to differentiate between a regular J and this new loop-merging operation, denoted J'.

If the boundary of the holes must be transmitted as part of the compressed data stream, the compression algorithm may order them as they are encountered by Edgebreaker. It may also start listing the vertices of each loop by the two vertices of the gate.

This new operation requires changing our coding scheme. A simple approach would be to use 4-bit codes for J and J' (for example, J could be 1000 and J' could be 1001. This solution adds |J| bits to the overall compressed representation even if there was a single hole. A different approach would be to encode J' operations with the same 3-bit code as J operations, but to identify them in a separate description that would precede O in the compressed format. J' operations may be identified by counting the number of J operations that separate them. A variable length integer format may be used to encode these counts.

If the boundaries of the holes do not need to be transmitted, the compression algorithm computes for each J':

- the count of J operations that separate it from its predecessor
- the identifier of the corresponding hole
- the identifier of the vertex in that hole

Note that the compression and the decompression algorithm know the total umber of holes and the total number of vertices in each hole. Consequently, the appropriate numbers of bits may be used to encode economically each one of these numbers.

E.5. Handles

When a J operation is performed as described earlier on a mesh where the current boundary wraps around a handle, a hole is created, instead of a separate component of the mesh. We use the technique described above for holes to deal with these cases. Basically, each time a hole is split, we need to mark the vertices with a new integer loop identifier so that when a J' operation reaches a loop, we may efficiently find out which loop it is and encode the two identifiers listed above for the J' operation.

F. Conclusion

We have presented a new compression/decompression technique for coding the triangle/vertex incidence graph of arbitrary triangle meshes. Our technique, called Edgebreaker, compresses the incidence of common meshes down to between 1.5 and 2 bits per triangle. By allowing additional bits, the basic technique may be extended to support more general triangle meshes and to allow in-line decompression. Edgebreaker's advantages over previously published approaches lies in its superior compression ratios, in the simplicity and efficiency of its compression and decompression algorithms, and in the fact that the incidence graph may be decompressed first and used to decompress vertices. Thus, Edgebreaker may be easily combined with a variety of geometry compression schemes based on vertex estimates that are derived from the incidence graph and from the location of previously decodedvertices. We have provided a careful analysis of the Edgebreaker approach and a detailed description of its implementation. We hope that these will help practitioners integrate the Edgebreaker technology in tools and standards for accessing 3D data over the internet.

G. Acknowledgments

This work has benefited from the generous equipment grants from the Intel and the IBM corporations. We would also like to thank Andrzej Szymczak from Georgia Tech for pointing out that the CL and CF combinations are impossible, Antonio Haro from Georgia Tech for developing a prototype implementation of the Edgebreakercompression and decompression algorithms, and Gabriel Taubin from IBM Research, and Leonard Schulman, Peter Lindstrom and Greg Turk from Georgia Tech for their discussions on coding schemes and on the general topic of geometric compression.

H. Bibliography

[Bar-Yehuda96] R. Bar-Yehuda and C. Grotsman, Time/space tradeoffs for polygon mesh rendering. ACM Transactions on Graphics, 15(2):141-152, April 1996.

[Carey97] R., Carey, G. Bell, C. Martin, The Virtual Reality Modeling Language ISO/IEC DIS 14772-1, April 1997, http://www.vrml.org/Specifications.VRML97/DIS.

[Darsa97] L., Darsa, B.Costa Silva, and A. Varshney, Navigating static environments using image-space simplification and morphing, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 7-16, April 1997.

[Deering95] M. Deering, Geometry Compression, Computer Graphics, Proceedings Siggraph'95, 13-20, August 1995.

[Denny97] M. Denny and C. Sohler, Encoding a triangulation as a permutation of its point set, Proc. Of the ninth Canadian Conference on Computational Geometry, pp. 39-43, Ontario, August 11-14, 1997.

[Dobkin85] D. Dobkin and D. Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, Journal of Algorithms, vol 6, pp. 381-392, 1985.

[Evans96] F. Evans, S. Skiena, and A. Varshney, Optimizing Triangle Strips for Fast Rendering, Proceedings, IEEE Vizualization'96, pp. 319--326, 1996.

[Heckbert97] P. Heckbert and M. Garland, Survey of Polygonal Surface Simplification Algorithms, in *Multiresolution Surface Modeling Course,* ACM Siggraph Course notes, 1997.

[Hoppe93] H. Hoppe, T, DeRose, T, Duchamp, J, McDonald, and W, Stuetzle, Mesh optimization, Proceedings SIGGRAPH'93, pp:19-26, August 1993.

[Hoppe96] H. Hoppe, Progressive Meshes, Proceedings ACM SIGGRAPH'96, pp. 99-108, August 1996.

[Hoppe97] H. Hoppe, View Dependent Refinement of Progressive Meshes, Proceedings ACM SIGGRAPH'97, August 1997.

[Kirkpatrick83] D. Kirkpatrick, Optimal search in planar subdivisions, SIAM Journal on Computing, vol **12**, pp. :28-35, 1983.

[Lee77] D.T. Lee and F.P. Preparata, Location of a point in a planar subdivision and its applications. SIAM J. on Computers, 6:594-606, 1977.

[Mann97] Y. Mann and D. Cohen-Or, Selective Pixel Transmission for Navigation in Remote Environments, Proc. Eurographics'97, Budapest, Hungary, September 1997.

[Mark97] W., Mark, L. McMillan, and G. Bishop, Post-rendering 3D warping, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 7-16, April 1997.

[Massey67] W. Massey, *Algebraic Topology: An Introduction,* Harcourt, Brace& World Inc., 1967.

[Naor90] M. Naor, Succinct representation of general unlabeled graphs, Discrete Applied Mathematics, vol. 29, pp. 303-307, North Holland, 1990.

[Neider93] J. Neider, T. Davis, and M. Woo, OpenGL Programming Guide, Addison-Wesley, 1993.

[Nelson89] M. R. Nelson, LZW Data Compression, Dr. Dobb's Journal, October 1989.

[Rockwood98] A. Rockwood, K. Heaton, and T. Davis, Real-time Rendering of Trimmed Surfaces, Computer Graphics, 23(3):107-116, 1989.

[Ronfard96] R. Ronfard. and J. Rossignac, Full-range approximation of triangulated polyhedra, Proc. Eurographics'96 , Computer Graphics Forum, pp. C-67, Vol. 15, No. 3, August 1996.

[Rossignac94] J. Rossignac, Through the cracks of the solid modeling milestone, *From Object Modelling to Advanced Visual Communication,* Eds. Coquillart, Strasser, Stucki, Springer-Verlag, pp. 1-75, 1994.

[Rossignac97] J. Rossignac, Geometric Simplification and Compression, in *Multiresolution Surface Modeling Course,* ACM Siggraph Course notes 25, Los Angeles, 1997.

[Snoeyink97] J. Snoeyink and M. van Kerveld, Good orders for incremental (re)construction, Proc. ACM Symposium on Computational Geometry, pp. 400-402, Nice, France, June 1997.

[Taubin98] G.Taubin and J. Rossignac, Geometric Compression through Topological Surgery, ACM Transactions on Graphics, Volume 17, Number 2, pp. 84-115, April 1998.

[Taubin98b] G.Taubin, W. Horn, F. Lazarus, and J. Rossignac, Geometry Coding and VRML, Proceedings of the IEEE, pp: 1228-1243, vol. 96, no. 6, June 1998.

[Turan84] G., Turan Succinct representations of graphs, Discrete Applied Math, 8: 289-294, 1984.

[Welch84] T Welch, A Technique for High-Performance Data Compression, Computer, June 1984.

[Ziv77] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, May 1977.

Jarek Rossignac is Professor in the College of Computing at Georgia Institute of Technology and the Director of GVU, Georgia Tech's Graphics, Visualization, and Usability Center, which involves 51 faculty members and over 160 graduate students focused on technologies that make humans more effective. Prior to joining Georgia Tech, he worked at the IBM T.J. Watson Research Center as the Strategist for Visualization; the Senior Manager of the Visualization, Interaction, and Graphics department; and the Manager of several IBM's graphics products: 3DIX, Data Explorer, and PanoramIX. His research interests focus on 3D geometric modeling and graphics, and on interactive and intuitive techniques for collaborative 3D design and inspection. He received numerous Best Paper and Invention awards, chaired 12 conferences, workshops, and program committees in Graphics, Solid Modeling, and Computational Geometry, guest edited 7 special issues of professional journals, and co-authored 13 patents. He holds a PhD in EE from the University of Rochester, New York in the area of Solid Modeling and a Diplome d'Ingenieur from the ENSEM in Nancy, France.